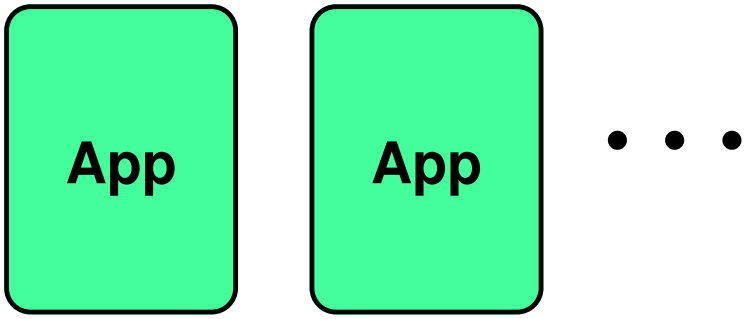
OS Components



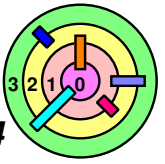
Applications

OS

Processor Management

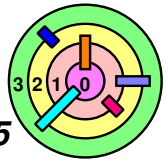
Memory Management

I/O Management

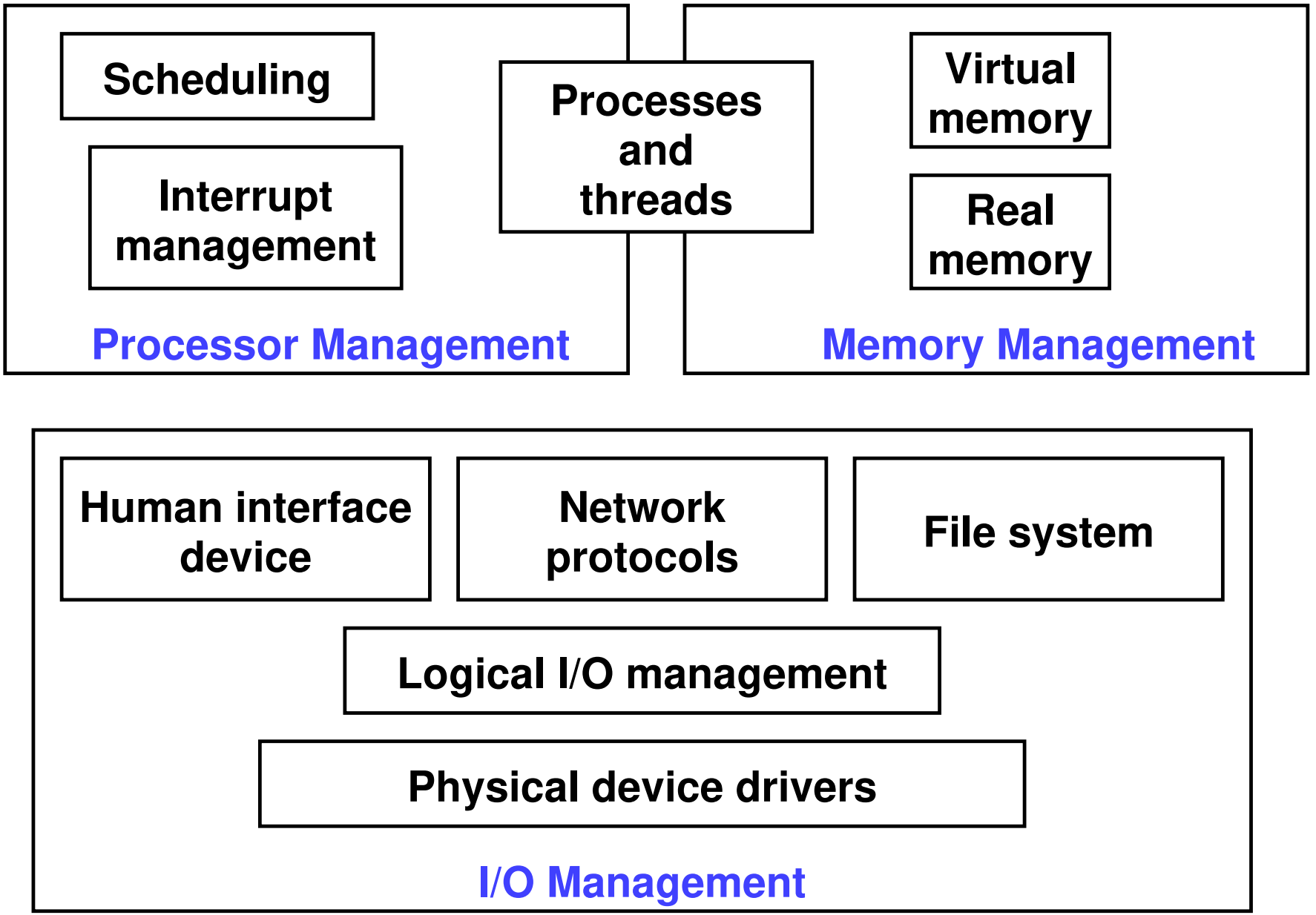


A Simple System: To Be Discussed

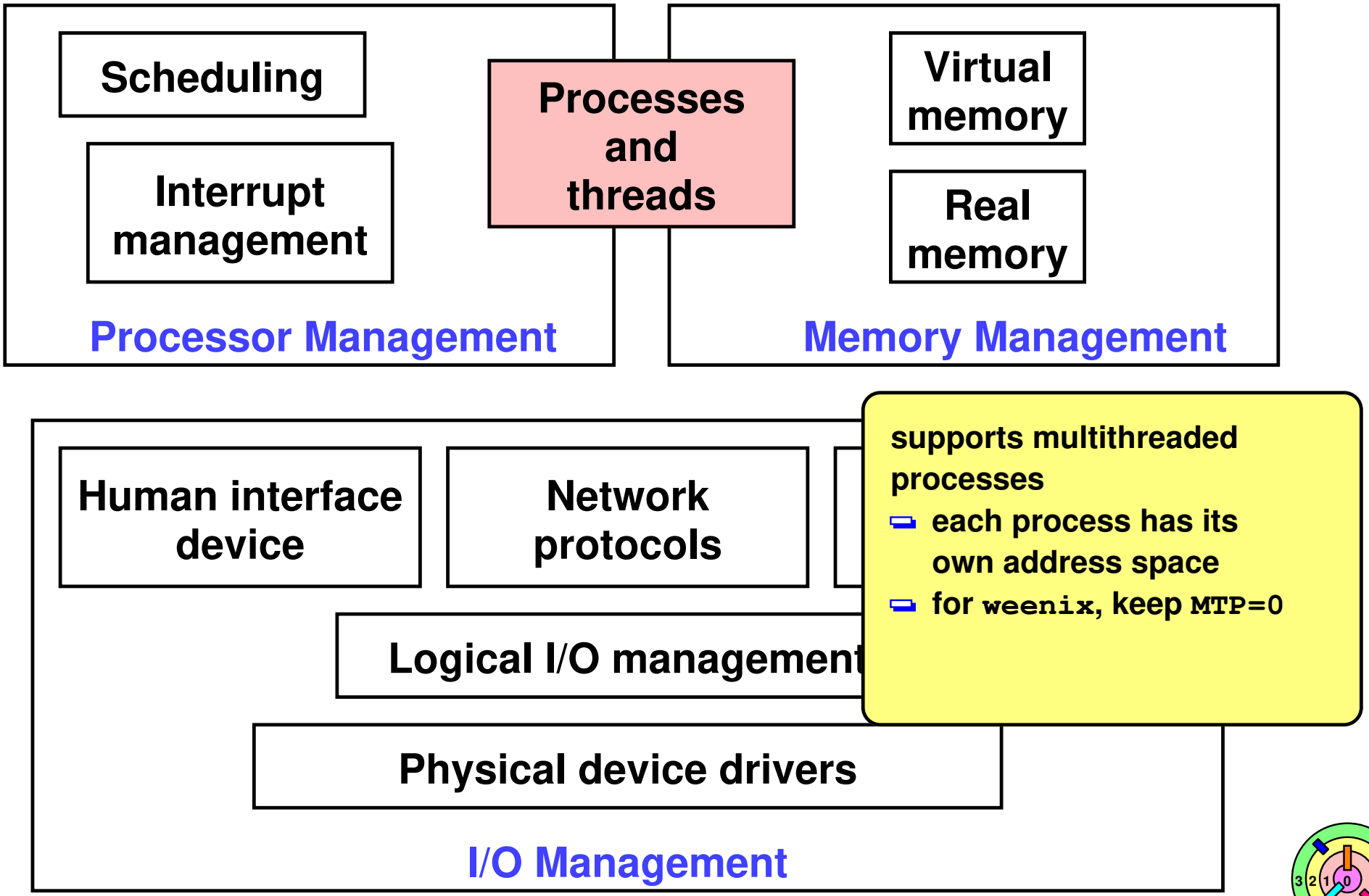
- What is the functionality of the components?
- What are the key data structures?
- What mechanisms are there to support the applications?
- How is the system broken up into modules?
- To what extent is the system extensible?
- What parts run in the OS kernel in privileged mode? What parts run as library code in user applications? What parts run as separate applications?
- In which execution contexts do the various activities take place?
  - e.g., thread context vs. interrupt context



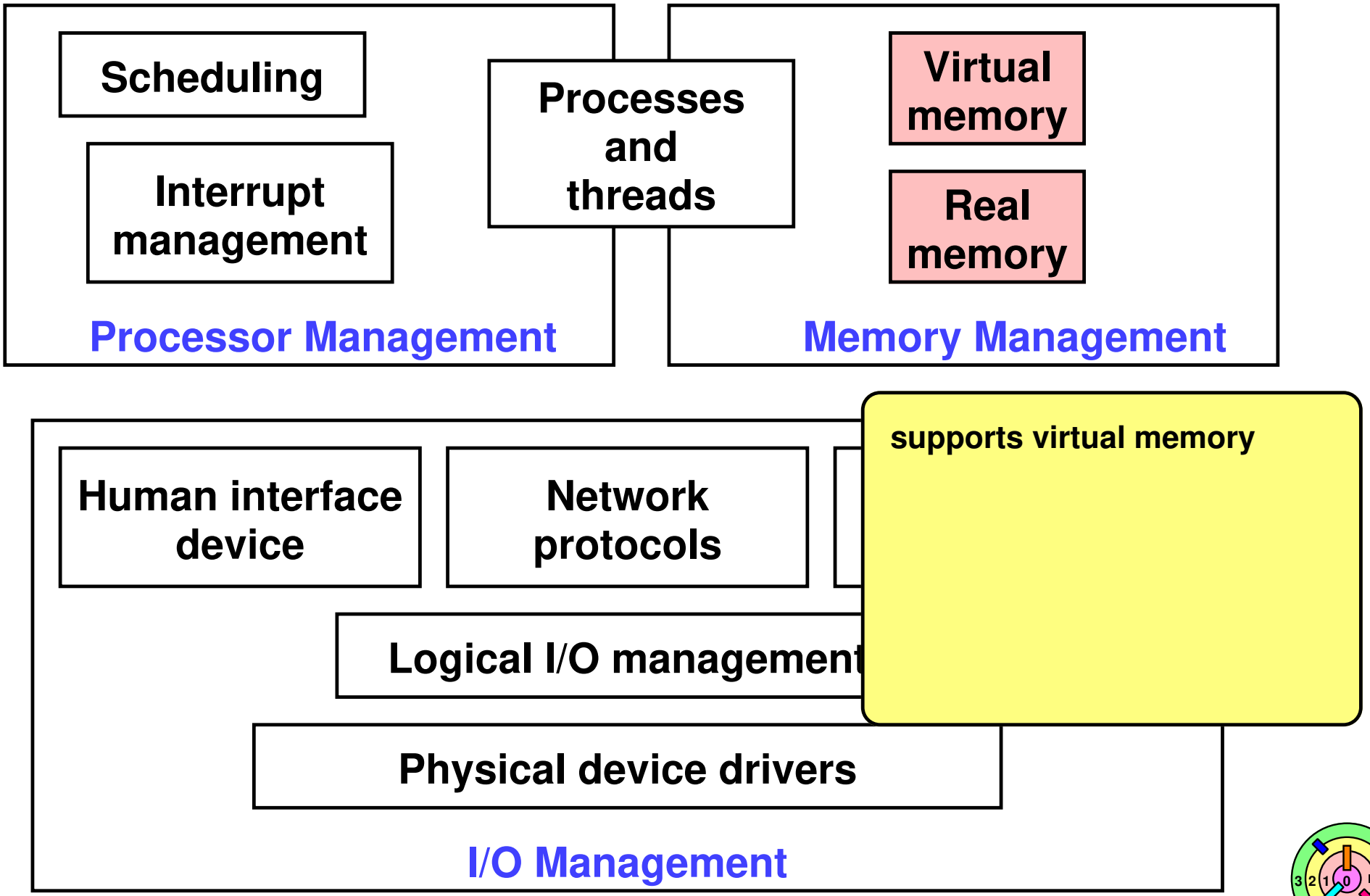
OS Components



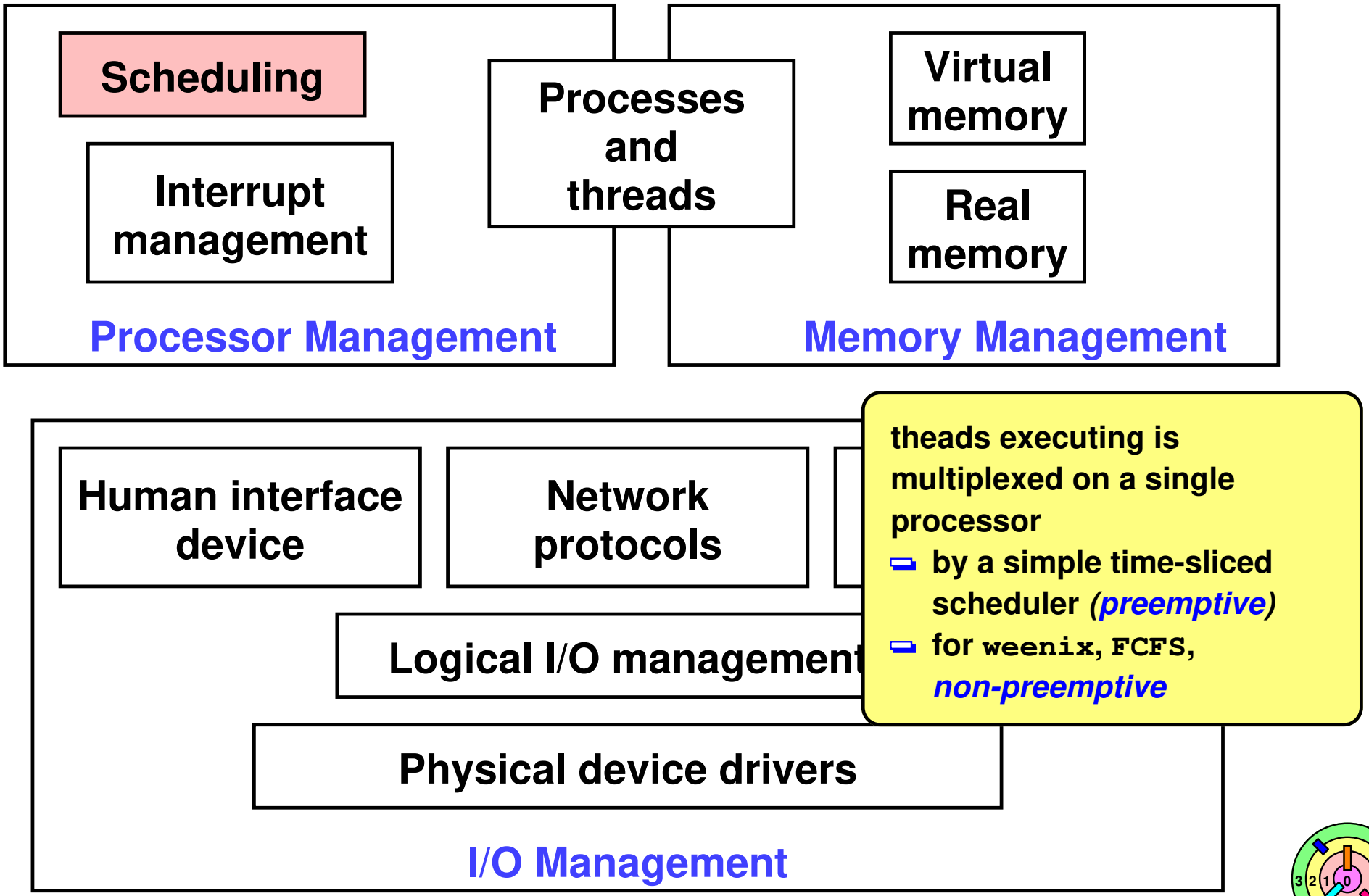
OS Components

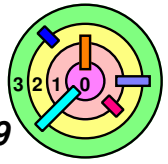


OS Components



OS Components





OS Components

has a file system
- layered on disks

Processes and threads

Virtual memory

Real memory

ment

Memory Management

Human interface device

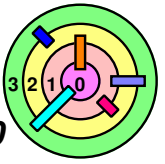
Network protocols

File system

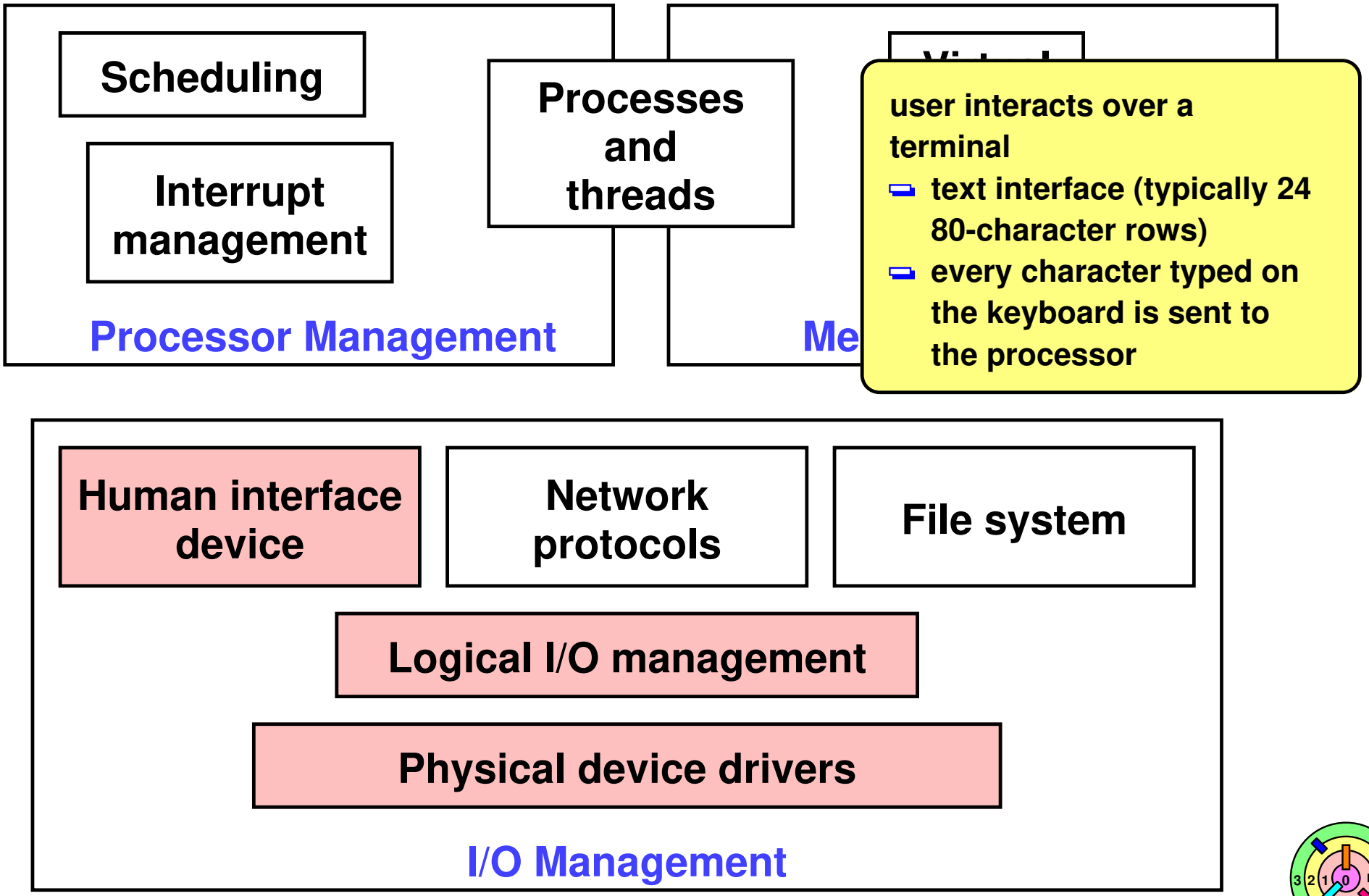
Logical I/O management

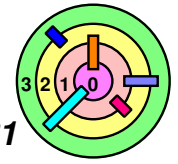
Physical device drivers

I/O Management

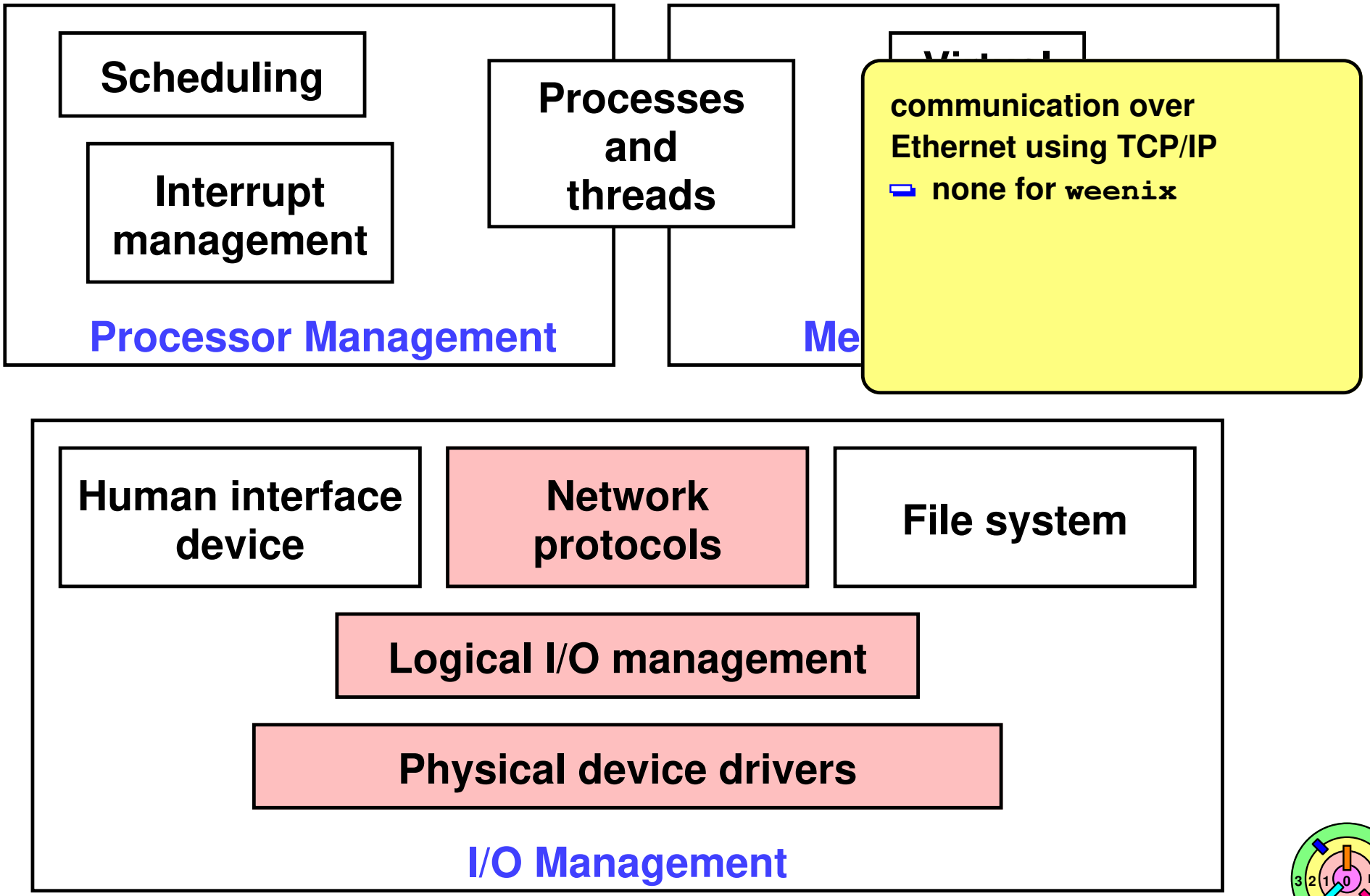


OS Components



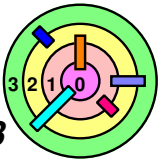


OS Components



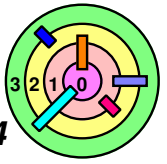
Some Important OS Concepts

- From an application program's point of view, our system has:
  - processes with threads
  - a file system
  - terminals (with keyboards)
  - a network connection
- Need more details on these... Need to look at:
  - how can they be provided
  - how applications use them
  - how this affects the design of the OS



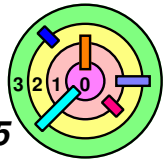
Processes And File Systems

- The purpose of a *process*
  - holds an *address space*
  - holds a group of *threads* that execute within that address space
  - holds a collection of references to *open files* and other *"execution context"*
- *Address space:*
  - set of addresses that threads of the process can usefully reference
  - more precisely, it's the content of these addressable locations
    - text, data, bss, dynamic, stack segments/regions and what's in them
      - a *memory segment/region* contains usable *contiguous* memory addresses



Address Space Initialization

- Design issue:
  - how should the OS *initialize* these address space regions?
- Unix does it in two steps
  - make a copy of the address space using `fork()`
  - then copy contents from the file system to the process address space (as part of the *exec* operation)
  - quite wasteful (both in space and time) for the text region since it's read-only data
    - should *share* the text region
  - what about data regions? they can potentially be written into
    - can also *share* a portion of a data region if that portion is never modified
    - copy data structures are much faster than copy data



Remember This?

Virtual Memory

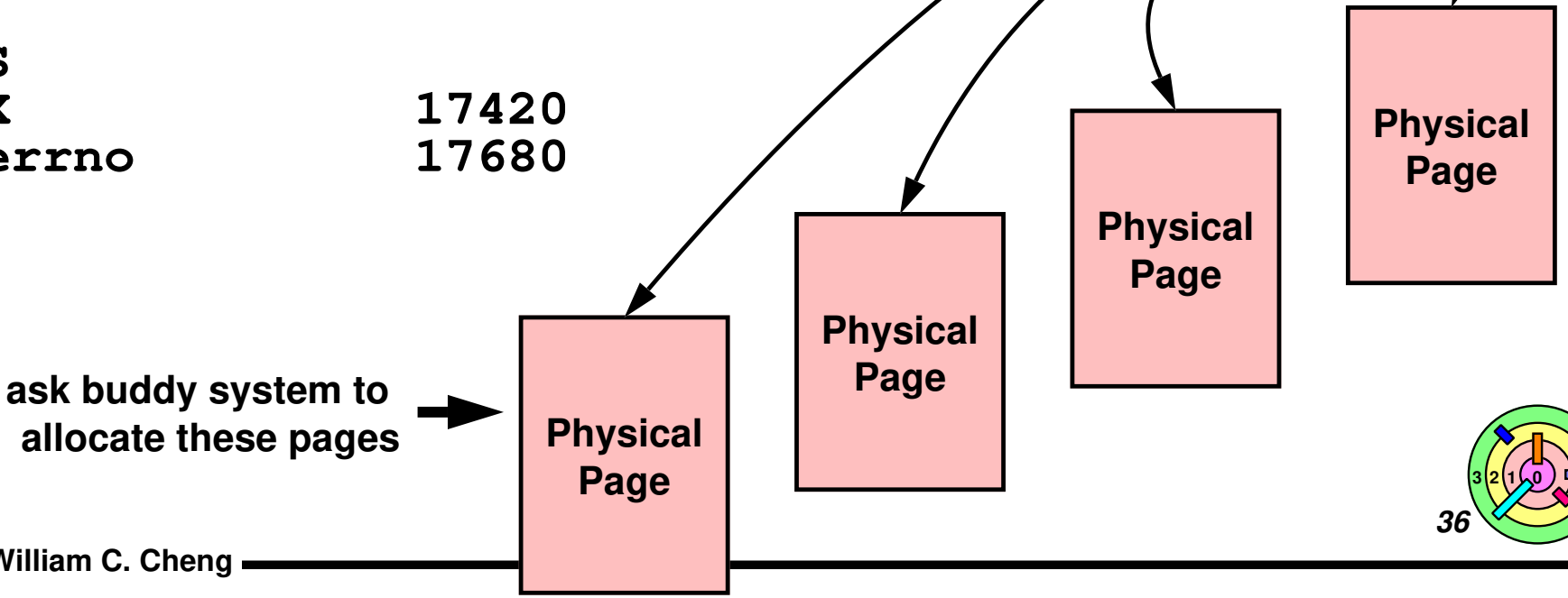
```
Text
   main              4096
   subr              4132
   printf            4156
   write            16156
   startup          16172

Data
   aX               16384
   printfargs       16388
   StandardFiles    16396

BSS
   X                17420
   errno            17680
```

Page Table

#	Start	Access	Physical Addr
0	0	-	-
1	4096	R	
2	8192	R	
3	12288	R	
4	16384	R/W	



Processes Can Share Memory Pages

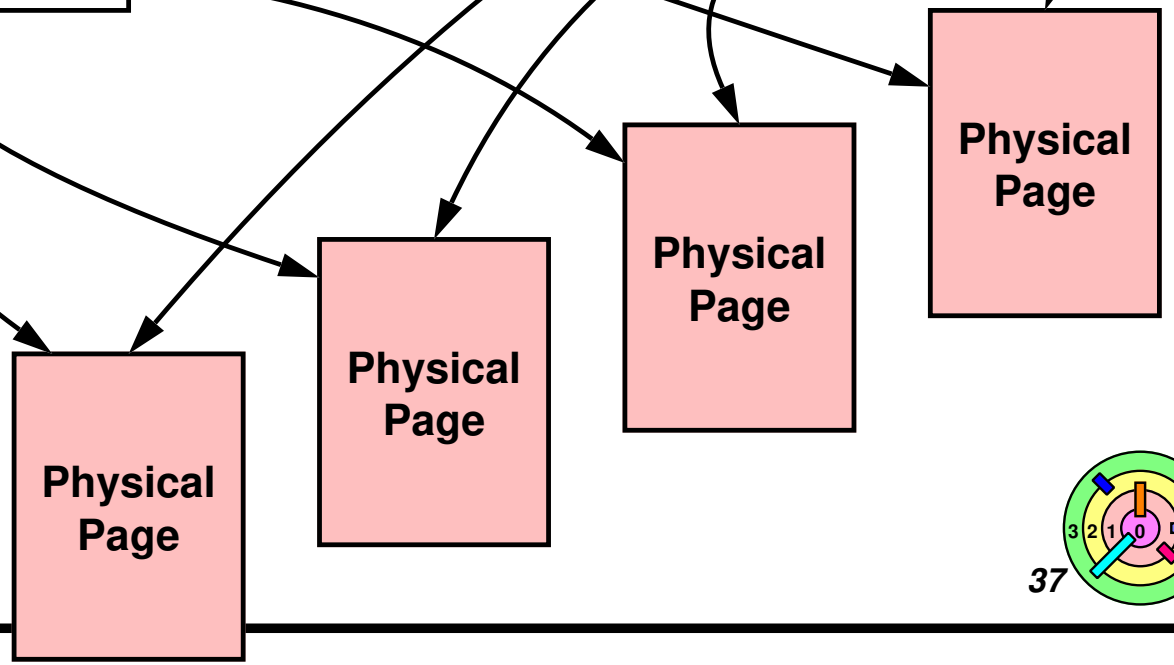
- Inside `fork()`, can simply copy parent's page table to child

Child Page Table

#	Start	Access	Physical Addr
0	0	-	-
1	4096	R	
2	8192	R	
3	12288	R	
4	16384	R/W	

Parent Page Table

#	Start	Access	Physical Addr
0	0	-	-
1	4096	R	
2	8192	R	
3	12288	R	
4	16384	R/W	



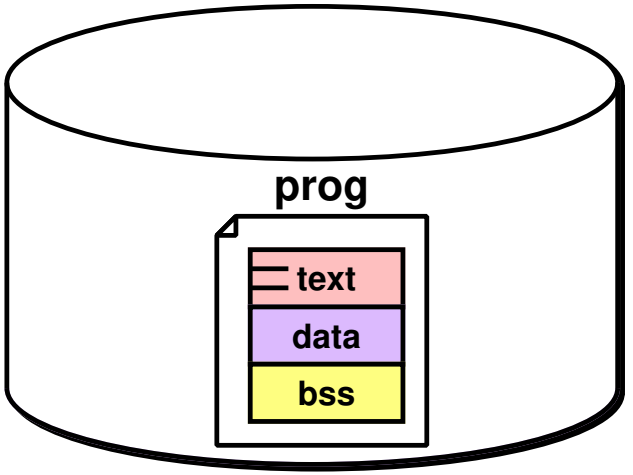
- power of *indirection*

exec()

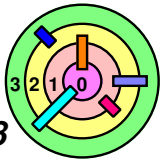
- Inside `exec()`, need to wipe out the address space (and page table) and create a new address space (and page table)

Child Page Table

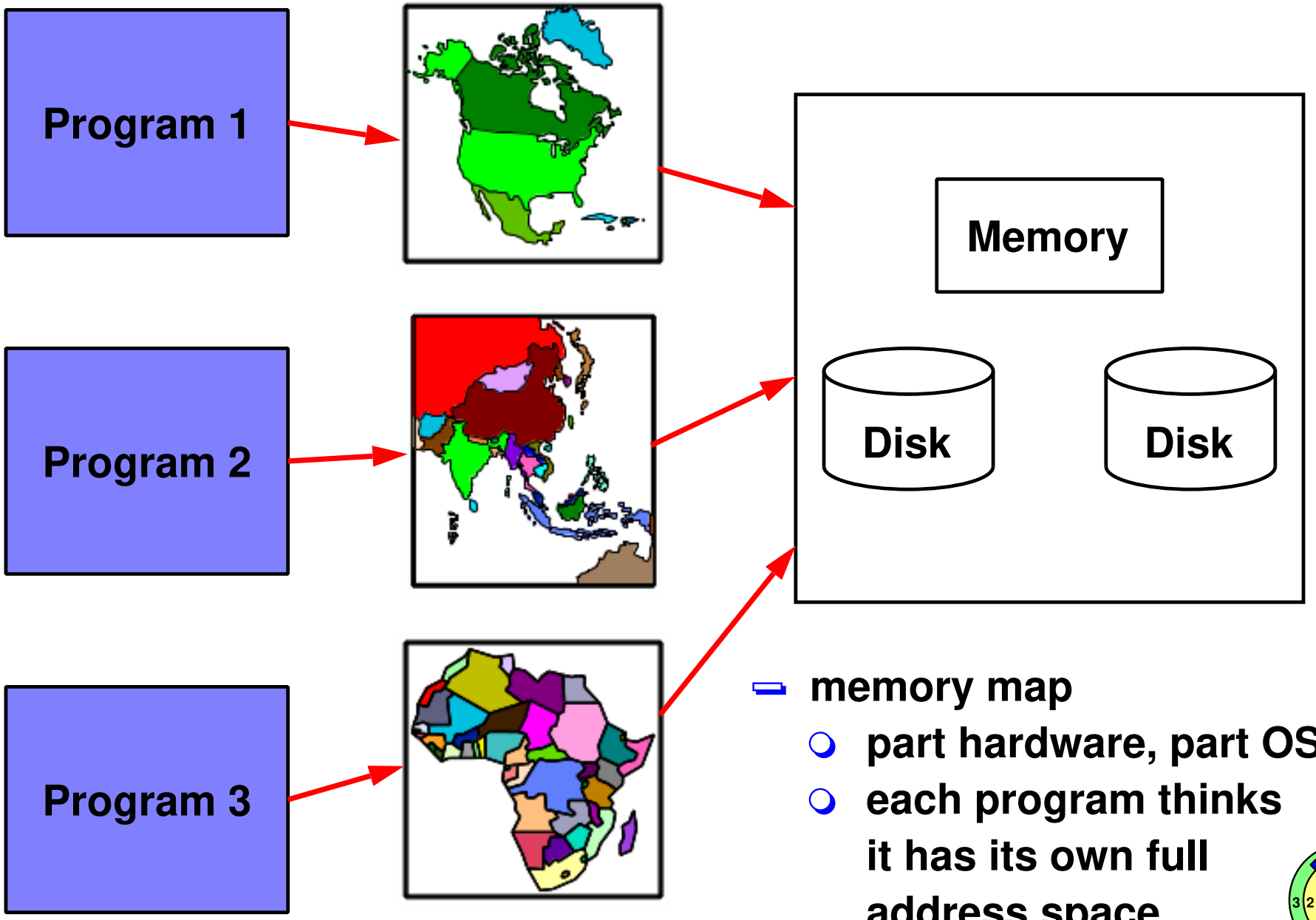
#	Start	Access	Physical Addr
0	0	-	-
1	4096	-	-
2	8192	-	-
3	12288	-	-
4	16384	-	-



- should you copytext and data segments of the new program from disk into memory now?
  - can be quite wasteful if you quit your new program quickly (and only use a small amount of the data you just copied form disk)



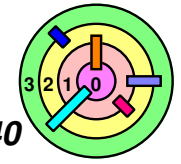
Memory Map



- memory map
  - part hardware, part OS
  - each program thinks it has its own full address space

Memory Map

- For the text region, why bother copying the executable file into the address space in the first place?
  - can just *map* the *file* into the *address space* (Ch 7)
    - *mapping* is an important concept in the OS
      - file mapping is *not* the same thing as address translation
      - some virtual memory pages map to files, and some map to physical memory
    - *mapping* let the OS *tie* the regions of the address space to the file system
    - address space and files are divided into pieces, called *pages*
    - if several processes are executing the same program, then at most one copy of that program's text page is in memory at once
  - *text regions* of all processes running this program are setup, using hardware address translation facilities, to share these pages
    - this type of mapping is known as *shared mapping*

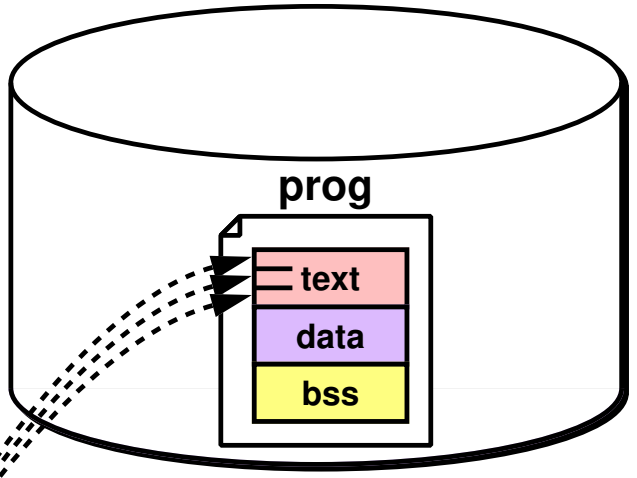


Memory Map

- The kernel uses a *memory map* to keep track of the mapping from *virtual pages* to *file pages*

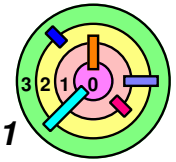
Child Page Table

#	Start	Access	Physical Addr
0	0	-	-
1	4096	-	-
2	8192	-	-
3	12288	-	-
4	16384	-	-

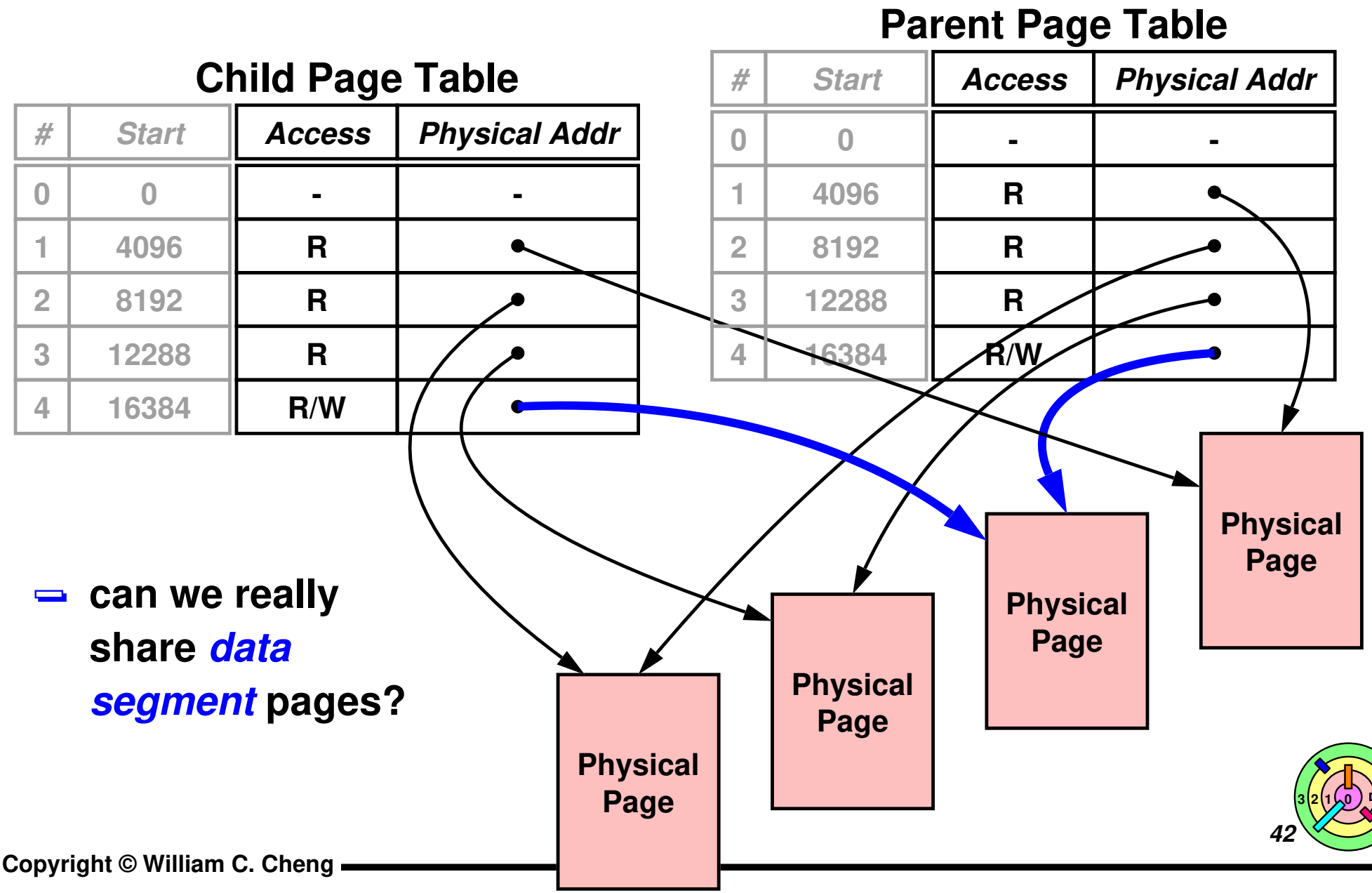


OS

- the kernel also uses *memory map* to keep track of the mapping from *virtual pages* to *physical pages*
  - also use it to maintain the page table data structure



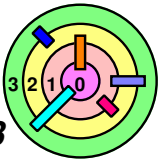
Processes Can Share Memory Pages



- can we really share *data segment* pages?

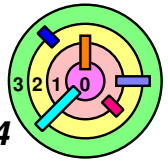
Address Space Initialization

- *Text regions* uses *shared mapping*
- *Data regions* of all processes running this program *initially* refer to pages of memory containing the *initial* data region
  - this type of mapping is known as *private mapping*
    - when does each process really need a private copy of such a page?
      - when data is *modified* by a process, it gets a *new* and *private* copy of the initial page



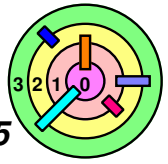
Copy-On-Write

- ***Copy-on-write (COW):***
  - a process gets a ***private*** copy of the page after a thread in the process performs a ***write*** to that page for the ***first time***
    - the basic idea is that only those pages of memory that are modified are copied
- Use private mapping and copy-on-write for data and bss regions
- The dynamic/heap and stack regions use a special form of private mapping
  - their pages are initialized, with zeros (in Linux); ***copy-on-write***
    - these are known as ***anonymous pages***
- If we can implement copy-on-write at the right time, then it's perfectly okay for processes to share address spaces
  - details in Ch 7



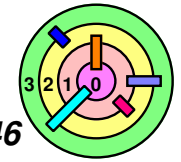
Shared Files

- If a bunch of processes *share* a *file*
  - we can also *map* the file into the *address space* of each process
  - in this case, the mapping is *shared*
  - when one process modifies a page, no private copy is made
    - instead, the original page itself is modified
    - everyone gets the changes
    - and changes are written back to the file
      - more on issues in Ch 6
- Can also share a file read-only
  - writing through such a map will cause segmentation fault



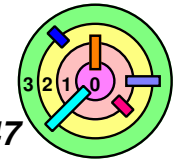
Memory Maps Summary

- File mapping
  - shared mapping
    - R/W: may change shared data *on disk*
    - R/O: read-only
  - private mapping
    - R/W: copy-on-write (will *not* change data on disk)
    - R/O: read-only
- Anonymous mapping
  - shared mapping (may be just shared with child processes)
    - R/W: may change shared data *in memory*
    - R/O: read-only
  - private mapping
    - R/W: copy-on-write
    - R/O: read-only
- Can also use all of the above in an application
  - `mmap()` system call



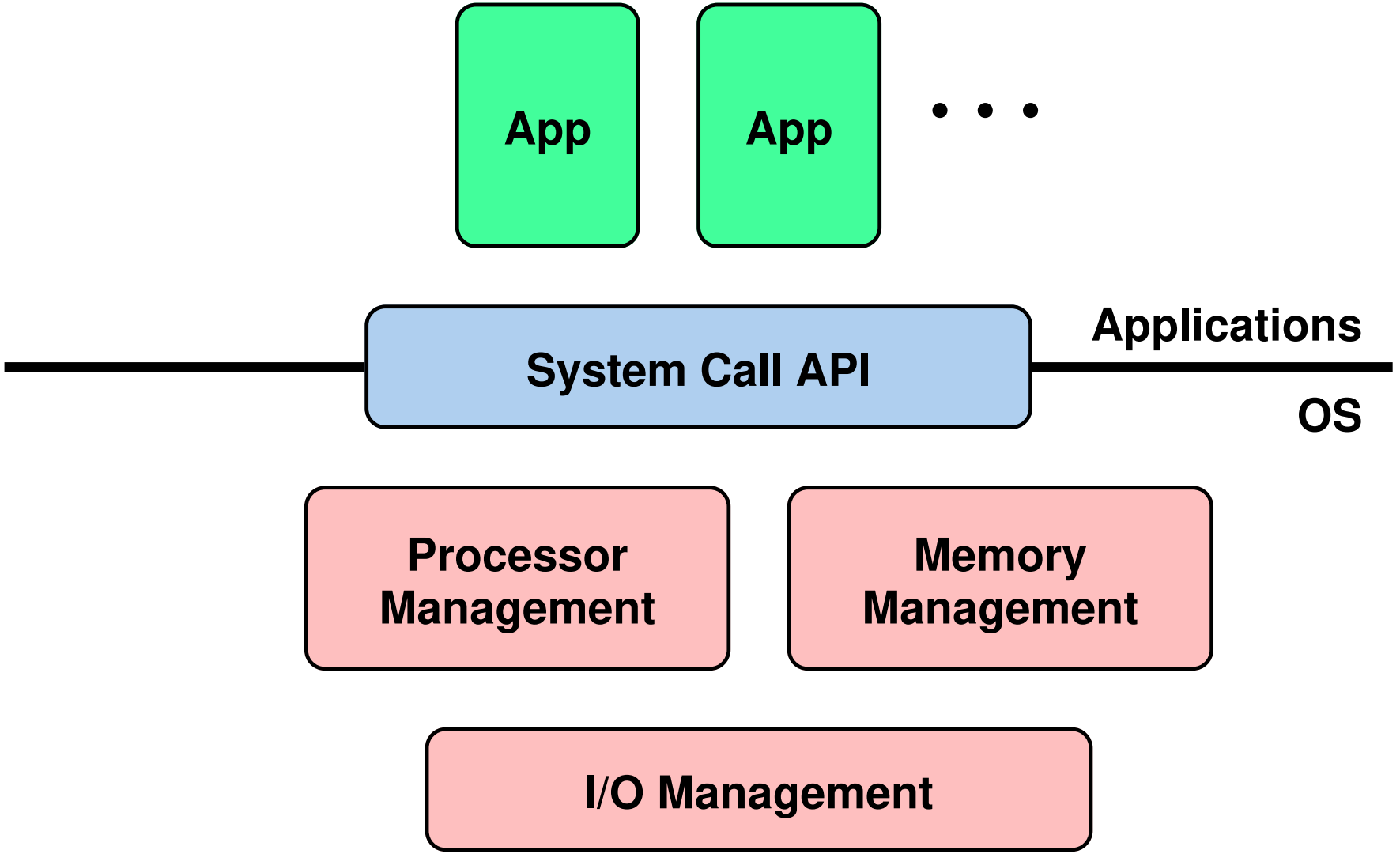
Block I/O vs. Sequential I/O

- Mapping files into address space is one way to perform I/O on files
  - block/page is the basic unit
  - some would refer to this as *block I/O*
- Some devices cannot be mapped into the address space
  - e.g., receiving characters typed into the keyboard, sending a message via a network connection
  - need a more traditional approach using explicit system calls such as `read()` and `write()`
  - this is referred to as *sequential I/O*
- It also makes sense to be able to read a file like reading from the keyboard
  - similarly, a program that produces lines of text as output should be able to use the same code to write output to a file or write it out to a network connection
  - makes life easier! (and make code more robust)



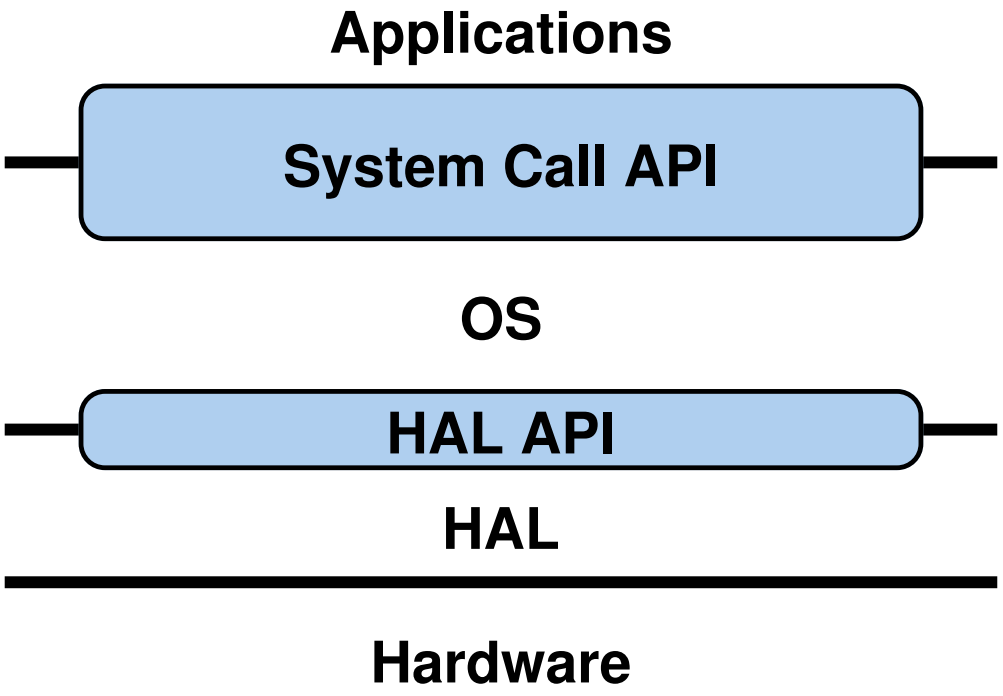
System Call API

- Backwards compatibility is an important issue
  - try not to change it much (to make developers happy)



Portability

- It is desirable to have a portable operating system
  - portable across various hardware platforms
- For a monolithic OS, it is achieved through the use of a *Hardware Abstraction Layer (HAL)*
  - a *portable interface* to *machine configuration* and *processor-specific operations* within the kernel



Hardware Abstraction Layer (HAL)

- Portability across machine configuration
  - e.g., different manufacturers for x86 machines will require different code to configure interrupts, hardware timers, etc.
- Portability across processor families
  - e.g., may need additional code for context switching, system calls, interrupting handler, virtual memmory management, etc.
- With a well-defined Hardware Abstraction Layer, most of the OS is *machine* and *processor independent*
  - porting an OS to a new computer is done by
    - writing new HAL routines
    - relink with the kernel